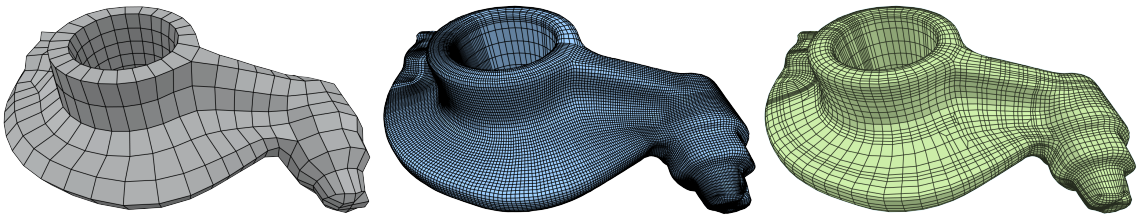
DOI: 10.1111/cgf.14076
Eurographics Symposium on Geometry Processing 2020
Q. Huang and A. Jacobson
(Guest Editors)

Volume 39 (2020), Number 5

Cost Minimizing Local Anisotropic Quad Mesh Refinement

M. Lyon[1], D. Bommes[2] and L. Kobbelt[1]

[1]RWTH Aachen University, Germany
[2]University of Bern, Switzerland



(a) *input* **(b)** *naive* **(c)** *ours*

Figure 1: *Reducing the approximation error by iteratively splitting all edges of the input model (**a**) whose distance to the original surface exceeds a threshold. Naively splitting these edges leads to a very dense mesh since local splits propagate globally along complete quad loops to preserve conformity (**b**). Our method places a few sparse singularities in order to better control the propagation flow of the splits leading to a much coarser mesh (**c**).*

Abstract

Quad meshes as a surface representation have many conceptual advantages over triangle meshes. Their edges can naturally be aligned to principal curvatures of the underlying surface and they have the flexibility to create strongly anisotropic cells without causing excessively small inner angles. While in recent years a lot of progress has been made towards generating high quality uniform quad meshes for arbitrary shapes, their adaptive and anisotropic refinement remains difficult since a single edge split might propagate across the entire surface in order to maintain consistency. In this paper we present a novel refinement technique which finds the optimal trade-off between number of resulting elements and inserted singularities according to a user prescribed weighting. Our algorithm takes as input a quad mesh with those edges tagged that are prescribed to be refined. It then formulates a binary optimization problem that minimizes the number of additional edges which need to be split in order to maintain consistency. Valence 3 and 5 singularities have to be introduced in the transition region between refined and unrefined regions of the mesh. The optimization hence computes the optimal trade-off and places singularities strategically in order to minimize the number of consistency splits — or avoids singularities where this causes only a small number of additional splits. When applying the refinement scheme iteratively, we extend our binary optimization formulation such that previous splits can be undone if this prevents degenerate cells with small inner angles that otherwise might occur in anisotropic regions or in the vicinity of singularities. We demonstrate on a number of challenging examples that the algorithm performs well in practice.

CCS Concepts

• ***Computing methodologies*** → ***Mesh models; Mesh geometry models;***

1. Introduction

Quad meshes have several advantages over triangle meshes that make them more desirable in industrial applications: They can naturally align to principal curvature directions and sharp features which is crucial in capturing shape features or the semantics of an object [BLP*12]. Other important applications are higher-order surface modeling, texturing and finite element simulation [BLP*12].

Thus, automatic generation of quad meshes, typically from a given triangle mesh, has been a well researched topic in the past decade [KNP07, BZK09, ZHLB10, ZHLB13, BCE*13, MPZ14, CBK15, JTPSH15, LHJ*15, HZN*18]. Some methods also offer control over quad sizing and may produce anisotropic meshes [PPTSH14, JFH*15, FBT*18]. While all these methods generate a quad mesh from scratch, it is sometimes necessary to start from a given quad mesh and only modify the mesh locally. For example

in an iterative design approach the user might be satisfied with one part of the mesh but not with another. In that case computing a new quad mesh from scratch in order to improve the unsatisfying region may lead to a less satisfactory result in what was previously a well structured region. It is thus necessary to develop algorithms which allow local mesh optimizations. The focus of this paper is the local mesh refinement, e.g. in order to increase the approximation quality in an adaptive simulation environment or provide more degrees of freedom to model fine detail features.

While there is previous work on local quad mesh modifications, only few of them target the refinement of quad meshes. Most existing quad mesh refinement algorithms either rely on splitting complete chords of quads which may affect large regions of the mesh, or they isotropically refine the mesh as local as possible disregarding the number of inserted singularities.

In this paper we derive a simple binary program formulation whose solution is a conforming anisotropic refinement of a given all-quad mesh (preserving the all-quad property) which is optimal with regard to a user-specified trade-off between number of singularities and number of mesh elements (Section 4). We extend the formulation in order to guarantee good minimal inner angles even when the algorithm is applied iteratively (Section 5). After briefly discussing a variation of our method that performs 3-refinement (Section 6) we show the results of our algorithm in a variety of different applications (Section 7).

2. Related Work

Daniels et al. [DSSC08] locally modify quad meshes in the context of coarsening. They define 3 coarsening operations: quadrilateral collapse, doublet collapse and poly-chord collapse. The former two are local, only affecting one and two quads respectively. The latter may affect large regions of the mesh if the chord forms a complex knot. Daniels et al. propose to use quadrilateral collapses first in order to simplify such long chords.

Tarini et al. [TPC*10] follow a similar approach to coarsen a quad mesh and define 6 local operations which all affect at most one element and its neighboring quads. In addition to collapse operations they define an edge rotation and a vertex rotation operation which locally changes the edge connectivity in order to improve the vertex valences.

An analysis of how singular vertices of a quad mesh can be (topologically) moved is done by Peng et al. [PZKW11]. In their paper they show that generating, moving and canceling of a single irregular vertex in an enclosed region is impossible. As an alternative they provide 3 operations to move pairs of singularities which they prove to be as local as possible, i.e. affecting the lowest number of quads possible.

Bommes et al. [BLK11] present an algorithm that aims to simplify the base complex of a given quad mesh. 2 simple operations, edge shift and edge collapse, which are local but used individually create non-quad elements, are combined to so called grid-preserving operators by finding a closed loop in a dual graph of the input. These grid-preserving operators change the topology of the base complex while maintaining an all quad mesh.

While all of the methods above perform local modifications to a given quad mesh, none of them deals with mesh refinement which is topic of this paper. Most closely related to our method is the the work presented by Schneiders [Sch96] who proposes to refine a mesh by assigning refinement levels to each vertex. Based on these refinement levels each quad is replaced by on of two templates where one template is a regular subdivision into four quads while the other allows a sizing transition. However, unlike our method the mesh is always refined as locally as possible without considering a possible reduction in generated singularities by additional regular refinement. Also Schneiders' refinement does not allow anisotropic refinement of quads which is possible with ours.

Anderson et al. [ABO09] present an algorithm for quad mesh adaptation based on quad chord and ring collapses as well as the refinement operations of Schneiders.

The αMST algorithm of Verma and Suresh [VS16] locally adapts a quad mesh by removing and completely remeshing the region of interest with as few singularities as possible. The resulting quadrangulation may be finer or coarser than the input.

With QuadMixer, Nuvoli et al. [NHE*19] locally modify quad meshes by replacing regions of one quad mesh with a region of another one. The structure of the input mesh patches remains unchanged with the exception of a transition region between the two parts.

Cherchi et al. [CAS*19] use a binary program formulation similar to ours in order to improve the element quality of a hexahedral mesh by inserting a padding layer. In contrast to our method their formulation cannot handle singular edges in the interior which limits the applicability to regular input models. Since the insertion of the buffer layer creates interior singularities their algorithm cannot be applied more than once to any given input

Tchon et al. [TDC04] present a simple scheme to anisotropically refine a quad mesh by iteratively shrinking layers of quads and inserting buffer layers around them, effectively splitting each edge into three. Compared to splitting edges into two, as in our case, this allows less control over the desired sizing. In addition, their algorithm does not consider the amount of inserted singularities.

3. Definitions

Let $Q = (\mathcal{V}, \mathcal{E}, \mathcal{H}, \mathcal{F})$ be a mesh with vertices $\mathcal{V}$, edges $\mathcal{E}$, halfedges $\mathcal{H}$ and faces $\mathcal{F}$. Q is a quad mesh if all faces are incident to exactly four halfedges.

In this paper we are interested in finding an optimal conforming refinement of a quad mesh. We define a refinement as a function over all edges $\varepsilon : \mathcal{E} \to \{0,1\}$ where $\varepsilon(e) = 1$ means that edge e is going to be split by inserting a vertex in the middle. A refinement is conforming if the number of edges which are split is even for every face as this allows replacing each face with a set of quads in a conforming manner.

4. Problem Formulation

Generating quad meshes typically amounts to finding a trade-off between accurately representing a target surface and keeping the number of mesh elements low.

Suppose a user defines a set $\mathcal{S}$ of edges which should be split, e.g. in order to decrease the Hausdorff distance to the original surface. A conforming refinement can always be found by splitting every quad strip in half that contains at least one edge in $\mathcal{S}$. However, these quad strips often run over large parts of the mesh leading to global mesh modifications. In the worst case, a single required split may lead to every quad being split into four. By placing singular vertices at appropriate positions the refinement can be kept local. However, since typically the number of singularities should also be low one has to find a compromise between number of elements and number of singularities added by the refinement.

In the following, we describe a method based on solving a binary program which finds the optimal refinement ε with regard to a user specified cost for singularities. For a detailed introduction to binary optimization, or more general integer optimization, we refer the interested reader to [NW88].

4.1. Refining a Single Quad

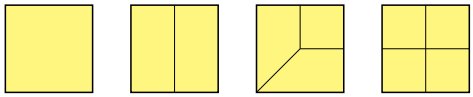


Figure 2: *Simple refinements of a single quad (**a**) after splitting two opposing edges (**b**), two adjacent edges (**c**) and all four edges (**d**).*

A single quad face has 4 edges, thus there are 2^4 possible constellations of edge splits that can be performed. However, since quad meshes always have an even number of boundary edges and an edge split increases the number of boundary edges by one the number of edge splits has to be even in order to enable remeshing the split face into a new quad mesh. Thus, the only valid split combinations are no splits, splitting two opposing edges (*1 to 2 split*), splitting two consecutive edges (*1 to 3 split*) and splitting all four edges (*1 to 4 split*). For these splits the face can be replaced by quads as shown in Figure 2. Note that in the case of splitting two adjacent edges an irregular vertex of valence 3 is inserted in the center of the face and for one of the original vertices the valence was increased by one, often resulting in a valence 5 vertex. Due to the shape of the inserted edges we call these *y-configurations*. These y-configurations can be used to transition between regions with different resolutions [Kob96].

4.2. Refining a Quad Mesh

When refining a quad mesh the considerations of the previous section hold for every individual quad of the mesh. Deciding which edges of each quad to split directly affects the choice of edges being split in the neighboring quads making the search for a conforming refinement a global problem. In addition to being conforming, the refinement should also be optimal with regard to the number of quads and singularities generated. While singularities are typically not desired they often allow for a refinement with a lot fewer edge splits and thus fewer generated quads.

We formulate the task of finding such an optimal set of edges that allow for a conforming refinement as a simple binary problem. For every edge e_i a binary variable s_i indicates whether the edge is split or not, which defines our refinement as $\varepsilon(e_i) = s_i$. One objective of our binary problem is to minimize the number of splits:

$$E_s = \sum_{e_i \in \mathcal{E}} s_i \tag{1}$$

To ensure, however, that at least all edges in the set of mandatory edges $\mathcal{S}$ are split we constrain the corresponding binary variable to 1:

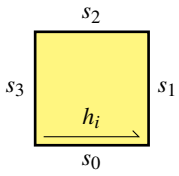
$$s_i = 1 \quad \forall e_i \in \mathcal{S} \tag{2}$$

As discussed in the previous section the number of split edges per face has to be even in order to allow a conforming refinement of the quad. Using two auxiliary binary variables a_i and b_i for every face f_i this property is encoded in the following constraints:

$$s_0 + s_1 + s_2 + s_3 = 2a_i + 4b_i \quad \forall f_i \in F \tag{3}$$

where s_0, s_1, s_2 and s_3 correspond to the four edges e_0, e_1, e_2, and e_3, of the face (cf. inset below).

Since y-configurations introduce undesired singularities we want to detect and penalize them. A y-configuration is used when two adjacent edges of a face are split while the other two are not. Thus, we add for every non-boundary halfedge $h_i \in \mathcal{H}$ a binary indicator variable t_i which we constrain to be 1 if and only if the two edges corresponding to h_i and the next halfedge in the same face are split and the others are not. This can be expressed with the following constraint:



$$t_i \geq 0.5 \cdot (s_0 + s_1 - s_2 - s_3) \quad \forall h_i \in \mathcal{H} \tag{4}$$

where s_0 to s_3 correspond to the four edges of the face in counter clockwise order starting at h_i. Note that this adds four constraints per face, one for each of the four rotationally symmetric y-configurations.

While constraint (3) ensures a conforming refinement, constraint (4) allows us to detect singularities generated by inserting y-configurations which are penalized by a user specified cost value $c_i \in [0, \infty)$:

$$E_y = \sum_{h_i \in \mathcal{H}} t_i \cdot c_i \tag{5}$$

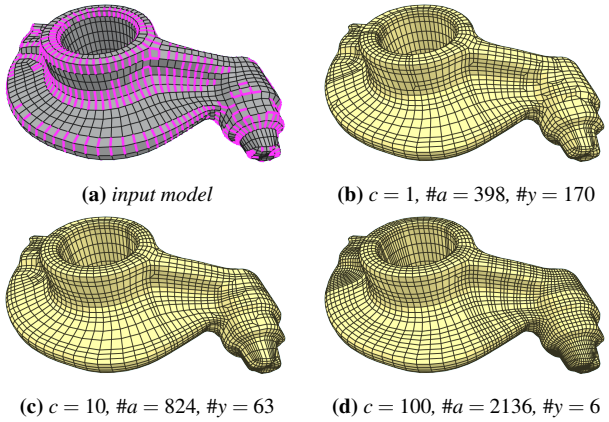
Putting the energies and constraints described above together yields our first binary program:

$$\begin{array}{ll} \text{minimize} & E = E_s + E_y \\ \text{subject to} & (2),\ (3),\ (4) \end{array} \tag{6}$$

The solution of this program is a valid refinement. Thus, each quad can be replaced by one of the templates depicted in Figure 2.

4.3. Costs for Singularities

There are multiple possibilities for choosing the singularity penalties c_i. In the most simple case of a constant penalty $c_i = c$ the solver will avoid inserting singularities unless inserting one allows



(a) *input model* **(b)** $c = 1$, $\#a = 398$, $\#y = 170$

(c) $c = 10$, $\#a = 824$, $\#y = 63$ **(d)** $c = 100$, $\#a = 2136$, $\#y = 6$

Figure 3: *Examples for different global singularity penalties $c_i = c$. #a is the number of splits added to the 1082 mandatory splits (magenta) in order to get a conforming refinement. #y is the number of inserted y-configurations.*

for a solution with c fewer splits. By increasing c the solution degenerates to the naive solution of splitting all quad strips which contain at least one mandatory split edge. Examples for different values of c are given in Figure 3.

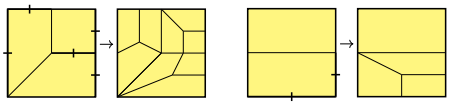
There are many other possible ways to design these costs depending on the intended application: A designer could mark regions in which singularities are penalized more or even not allowed at all such that they are only placed in regions of low importance. An example for this is given in Figures 8 and 9. Furthermore, geometric properties of the mesh such as the curvature could be used to promote singularities in flat regions rather than in curved regions which is beneficial for spline fitting applications. Finally, since we use one indicator variable t_i per halfedge, differently rotated y-configurations in the same face can be penalized differently based on which vertex the diagonal is connected to. This allows, for example, choosing a penalty depending on the angle at or the valence of that vertex.

5. Stability

Often it is not enough to split edges in half once and the algorithm has to be applied iteratively. However, successive refinement can lead to quads with small inner angles when y-configurations are split repeatedly (Figure 4a) as well as when y-configurations are inserted in narrow quads (Figure 4b). For many applications small angles are undesirable and thus require a refinement to be *stable*. A refinement is stable if the minimum angle does not depend on the refinement level [Sch96]. In the following, we address how stability is achieved by adapting our simple problem formulation from above.

5.1. Refinement of Y-Configurations

As shown in Figure 4a further refinement of y-configurations may lead to low quality quads. We therefore do not allow refinement of a y-configuration directly. Instead, if refinement is desired, the y-configuration is first replaced by a regular grid of four



(a) **(b)**

Figure 4: *Example of low quality faces created by iteratively refining y-configurations (**a**) and inserting a y-configuration into an anisotropic quad (**b**).*

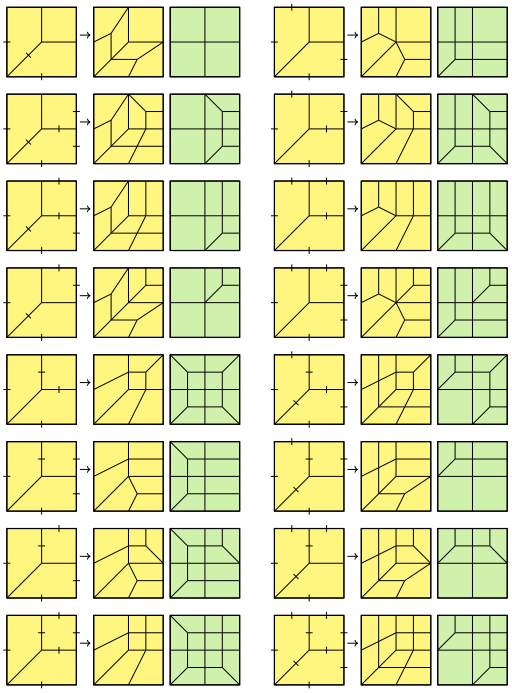
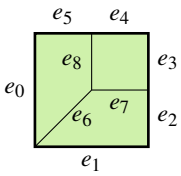


Figure 5: *All valid splits for a y-configuration where at least the two longer edges are split. In each block of three the left mesh shows with small ticks which edges are split. The mesh in the middle is the result of applying the simple replacement rules from Section 4 to every individual quad. Note that many edges are not aligned with the main directions. The right mesh is an alternative template for replacing the y-configuration where again all edges except for the diagonal edges of the new y-configurations are perfectly aligned.*

quads. For this to be possible the two long edges (e_0 and e_1 in the inset) have to be split. In practice, replacing the y-configuration with four quads and further refining these is done in one step by directly replacing the y-configuration with one of the 16 templates shown in Figure 5.



Note that Figure 5 lists all conforming refinements of the faces of a y-configuration where the two long edges are split. Thus, in order to restrict the solution of the binary problem to those 16 configurations we only need the following two constraints: Either none or both of the two long edges are split:

$$s_0 = s_1 \quad \forall y \in \mathcal{Y} \tag{7}$$

where $\mathcal{Y}$ is the set of y-configurations and s_0 and s_1 refer to the edges e_0 and e_1 of the inset above. The second requirement is that if any of the smaller edges is split, then the long edges have to be split as well:

$$\sum_{i=2}^{8} s_i \leq 7\, s_0 \quad \forall y \in \mathcal{Y} \tag{8}$$

These constraints can simply be added to the constraints of Problem 6 after the first iteration. However, the current energy will penalize refinements of y-configurations unnecessarily: For example, the first split configuration in Figure 5 will increase the cost because two of the t_j variables indicating the insertion of singularities will be set to 1 even though the y-configuration is eventually replaced by regular quads, actually reducing the number of singularities. In order to correctly model the costs we need to identify which of the 16 possible configurations is used to replace the y-configuration.

For each y-configuration $y_i \in \mathcal{Y}$ we add 16 auxiliary variables z_i^j, where $z_i^j = 1$ indicates that the jth configuration from Figure 5 is used to replace the corresponding y-configuration y_i. For each of these variables we add two constraints, one lower and one upper bound, which enforce that the variable is set to 1 if and only if the split variables s_i are set accordingly. Here we only give one example for the first configuration as a representative, a complete list of the 16 constraints and a derivation is given in the appendix:

$$0 \leq s_0 + s_1 - s_2 - s_3 - s_4 - s_5 + s_6 - s_7 - s_8 - 9z_i^1 + 6 \leq 8 \quad \forall y \in \mathcal{Y} \tag{9}$$

Using these constraints the energy is adjusted by multiplying the indicator variables z_i^j with a cost d_i^j which may be negative to reward the removal of the singularities:

$$E_\mathrm{a} = \sum_{y_i \in \mathcal{Y}} \sum_{j=1}^{16} z_i^j \cdot d_i^j \tag{10}$$

Notice that only one z_i^j can be one for every i due to the above constraints. For constant costs c_i we provide a list of corresponding costs d_i^j in the appendix.

5.2. No Y-Configurations in Narrow Faces

As already mentioned above, inserting a y-configuration into a very thin quad may introduce small angles, cf. Figure 4b. Since these small angles may cause numeric instabilities or other problems in many applications we want to avoid them. Therefore, in this section we first define the set of faces in which y-configurations may be inserted and then add a constraint preventing y-configurations for all other faces.

Each original quad defines a simple bilinear parametrization of its inside by mapping each corner to one of the four points $(0,0)$, $(1,0)$, $(1,1)$ and $(0,1)$. Each new vertex added by splitting an edge is inserted at the center of that edge. Vertices added inside a face, i.e. when a quad is replaced by four quads or a y-configuration, are inserted in the barycenter of the four vertices of the face.

Each new edge is a straight line within this parametrization domain. We define $l(e)$ to be the length of edge e in that domain. We can then define the set of faces which are parametric squares as those with edges with identical lengths:

$$\mathcal{F}_\square = \left\{ f \in \mathcal{F} \mid \forall e_i, e_j \in f : l(e_i) = l(e_j) \right\} \tag{11}$$

To guarantee good angles we only allow the insertion of y-configurations in those faces:

$$t_i = 0 \quad \forall f_i \in \mathcal{F} \backslash \mathcal{F}_\square \tag{12}$$

Adding the constraints and energy terms discussed in this section to the previous binary program yields our stable formulation:

$$\begin{array}{ll} \text{minimize} & E_\mathrm{s} + E_\mathrm{y} + E_\mathrm{a} \\ \text{subject to} & (2),\ (3),\ (4),\ (8),\ (9),\ (12) \end{array} \tag{13}$$

Note that even with the additional constraints the problem is still guaranteed to be feasible since splitting all edges is always a solution. Of course, typically solutions with a lot fewer splits can be found.

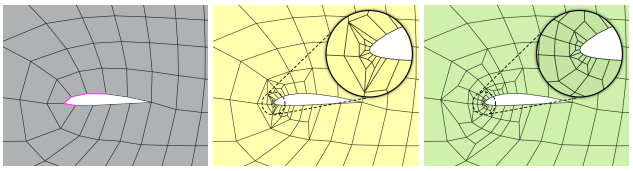


Figure 6: *Comparison of refining the contour of a wing profile (left) using our simple (middle) and our stable formulation (right).*

Figure 6 shows an example of contour approximation of a wing profile where boundary edges next to valence 3 vertices are split if the angle between them is smaller than a threshold. The repeated splitting of y-configurations leads to small angles (middle). Our stable formulation replaces y-configurations by a regular refinement before further refinement leading to better angles but a slightly less local refinement as an additional layer of transition elements is used.

6. 3-Refinement

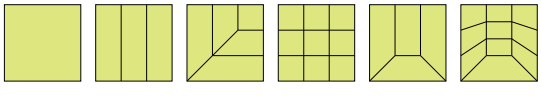


Figure 7: *Simple refinements of a single quad with at most one split per edge. Refining edges of a single quad into three parts can be achieved with similar patterns as discussed in Sections 4 and 5 when the number of split edges is even (**a**)-(**d**). In addition it enables a configuration where only one side (**e**) or three sides (**f**) are split.*

So far we have discussed dyadic splitting operations that split a single edge into two edges. These types of splits are sometimes

referred to as 2-refinement [Sch96]. In contrast, 3-refinement operations split edges into three edges. While 2-refinement is preferable due to the finer control over the increased mesh density, 3-refinements has the advantage of allowing not only templates replacing quads with an even number of split edges (cf. Figure 2) but also templates replacing quads with one or three split edges as shown in Figure 7.

This simplifies the search for a conforming set of split edges as now every set of edges is consistent and splitting only the mandatory split edges yields the result with the smallest number of splits. However, this result may include many undesirable singularities which could be reduced by splitting additional edges. An optimal trade-off between number of split edges and added singularities can again be found by solving a binary problem.

Our basic binary problem formulation (Problem (6)) is surprisingly easy to adapt for 3-refinement by simply dropping constraint (3):

$$\begin{array}{ll} \text{minimize} & E = E_s + E_y \\ \text{subject to} & (2),\ (4) \end{array} \tag{14}$$

Without that constraint the solver is not required to find an even number of split edges per face. Note however that with constraints (4) still in place, quads with a single split or three splits are penalized twice as much as a y-configuration which is the desired behavior since two valence 3 vertices are inserted and the valence of two of the original vertices is increased by one.

We leave stability considerations for future work. We conjecture that a formulation analogous to the one discussed in Section 5 can be used.

7. Results

We apply our method to a variety of different inputs. If not stated otherwise we use our stable formulation (Equation (13)) with a constant singularity penalty c_i. All quad meshes used in this paper are created from triangle meshes using the algorithm of Campen et al. [CBK15]. Of course, our algorithm does not depend on that specific algorithm and quad meshes from any other source could be used as well. So far we avoided a discussion about the placement of the newly inserted vertices as there are a plethora of options from simply placing the vertex in the center of the split face or edge to carefully optimizing the position based on application specific quality criteria. For simplicity we split edges and faces in the center and project the vertices in normal direction (3D examples) or use the parametrization provided by the algorithm of Campen et al. (2D examples) to map the vertices into the original surface. The binary problems are solved using the Gurobi solver [GO16] using default parameters (in particular with an optimality gap of 0). Statistics are summarized in Table 1.

In Figure 8 a region around the three airfoils has been specified in which the insertion of singularities has a high cost while in the outer regions the cost is low. Splitting all edges on the contour and those orthogonal to the contour then leads to a high resolution mesh without added singularities close to the airfoils.

A similar approach was used in Figure 9 where some edges (ma-

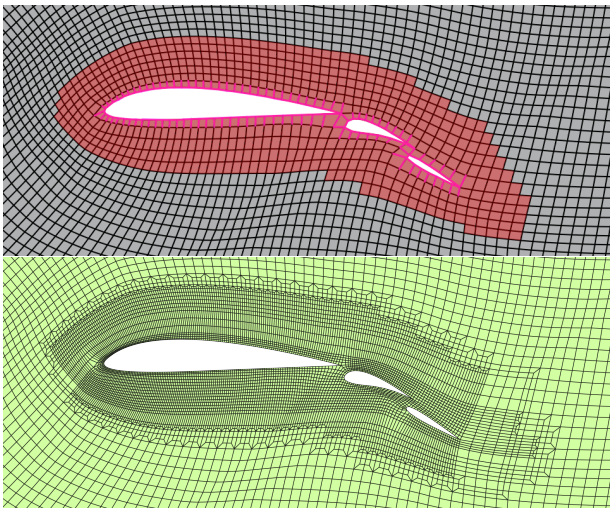


Figure 8: *High singularity cost in the marked region. Two iterations of splitting contour edges and edges orthogonal to contour area applied.*

Mesh	Method	$\lvert\mathcal{F}\rvert$	#y	time
Rockerarm (1)	ours	10108	48	5 s
Rockerarm (1)	naive	31472	0	< 1 s
Rockerarm c=1 (3)	ours	4344	170	21 s
Rockerarm c=10 (3)	ours	4819	63	21 s
Rockerarm c=100 (3)	ours	6799	6	< 1 s
Airfoil (8)	ours	15225	317	36 s
Hand (9b)	ours	16058	26	2 s
Hand (9c)	ours	17138	28	4 s
Hand (9d)	ours	17994	26	3 s
Fertility (11)	ours	2241	85	6 s
Fertility (11)	naive	4517	0	< 1 s
Roof (12)	ours	976	80	< 1 s
Fandsik (10)	ours	9014	129	7 s
Fandisk (10)	naive	33082	0	< 1 s
Pantasma (13)	[Sch96]	2353	210	< 1 s
Pantasma (13)	[TDC04]	2013	156	< 1 s
Pantasma (13)	ours	1317	39	3 s
3wayB (13)	[Sch96]	5394	627	< 1 s
3wayB (13)	[TDC04]	3896	356	< 1 s
3wayB (13)	ours	3251	46	6 s
Bot Shoulder (14)	[TDC04]	6162	356	< 1 s
Bot Shoulder (14)	ours (3)	6848	46	2 s
Octopus (14)	[TDC04]	14084	510	< 1 s
Octopus (14)	ours (3)	15262	162	24 s
Cylinder (15)	ours	356	18	587 s
Cylinder (15)	ours (3)	379	20	< 1 s

Table 1: *Statistics for our results showing the number of resulting elements $\lvert\mathcal{F}\rvert$ and inserted y-configurations #y, and the time of the algorithm. With ours (3) we refer to the formulation that uses 3-refinement (Section 6).*

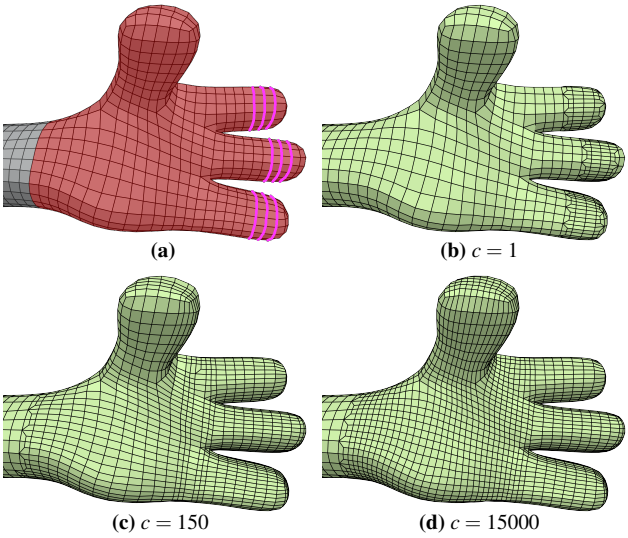


Figure 9: *Splitting the edges highlighted in magenta for different singularity costs c in the marked region.*

genta) on the finger tips are mandatory splits with varying singularity penalties in the highlighted region and low penalty on the rest of the body. Using a low penalty of $c = 1$ in the highlighted region leads to y-configurations directly next to the refined region in order to do a sizing transition as locally as possible. Choosing a very high penalty moves all singularities away from the hand to the wrist area. However, due to the structure of the quad mesh many quads in the palm of the hand have to be split in both directions. By setting a compromise penalty of 150 our method strategically places only two y-configurations within the highlighted region, significantly reducing the number of split edges.

In Figure 10 we show the results of iteratively splitting all edges orthogonal to the feature edges marked in magenta. Naive splitting increases the number of elements from 4.4k to 33k compared to 9k with our method.

In the previous examples the mandatory splits were all in one region and the goal of the optimization was to find a conforming refinement which only locally changes the mesh. In the following we discuss a different use case: reducing the Hausdorff distance between quad mesh and original surface. In that case the set of mandatory split edges is often distributed over the quad mesh and the optimization goal is to find a good trade-off between a low number of splits and a low number of singularities.

Since all vertices of the quad mesh already lie within the original surface we approximate the Hausdorff distance by measuring the distance between edge centers and their projection into the original surface along the average normal of the incident faces. We define the set of mandatory split edges as all edges for which that distance is larger than a given threshold.

In Figure 1, we compare the naive splitting of all quad loops which contain at least one mandatory split edge with our method using a constant singularity penalty of 50. Both methods fulfilled the Hausdorff distance threshold after three iterations. For the last

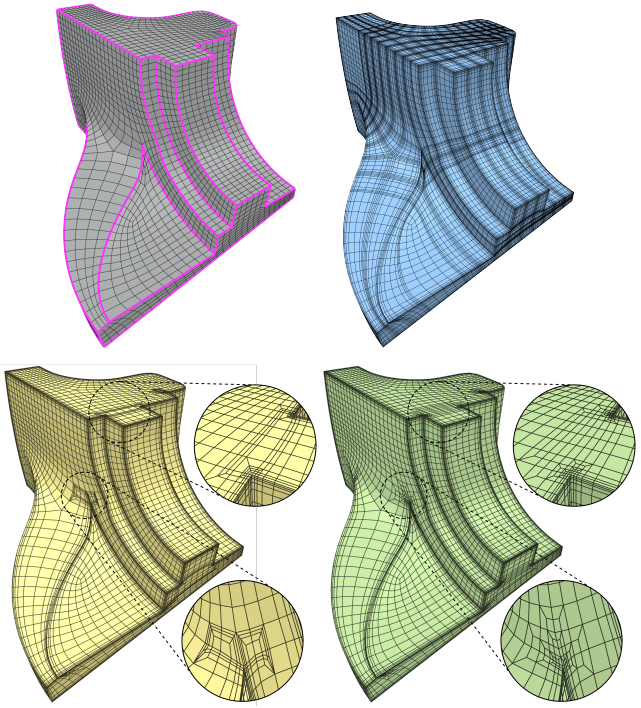


Figure 10: *Splitting edges of the fandisk which are orthogonal to the feature edges marked in magenta (top left). Note that using our non-stable formulation (Equation (6)) vertices with valences up to 7 and faces with very small inner angles are generated by iteratively splitting faces of y-configurations (bottom left). With our stable formulation (Equation (13)) all quads are well shaped due to using templates from Figure 5 to replace y-configurations (bottom right). The naive solution is shown in blue (top right).*

iteration the input model has 6.7k edges of which 506 need to be split. While the naive solution splits an additional 15015 edges, our formulation inserts 48 y-configurations requiring only 2595 additional edges to be split to form a conforming refinement.

Figure 11 shows another example of minimizing the Hausdorff distance on the fertility model. Due to the coarse base complex of the input the naive solution does not produce as fine a solution as on the rockerarm example. However, the naive formulation still increases the number of elements from 537 to 4.5k while our solution produces only half as many elements.

For Figure 12 we perform edge splits based on a physics based simulation of a flat roof supported on all four sides under the gravitational force of its own weight. The mesh is computed using the method of Lyon et al. [LCBK19] such that the edges are aligned with the two principal stress directions. For each edge the orthogonal stress is integrated to estimate how much load the four orthogonal edges have to bear. Edges for which this value exceeds a threshold are set as mandatory splits. In addition, splits are prevented (by constraining the corresponding variable to 0) for all edges incident to non-quad faces. The resulting refinement adds additional edges such that large forces are better distributed over more edges lead-

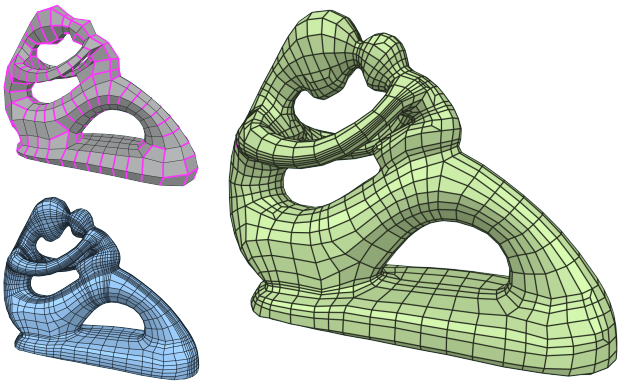


Figure 11: *Splitting all edges of Fertility (top) whose Hausdorff distance exceeds a threshold using naive splits (bottom) and our method (right).*

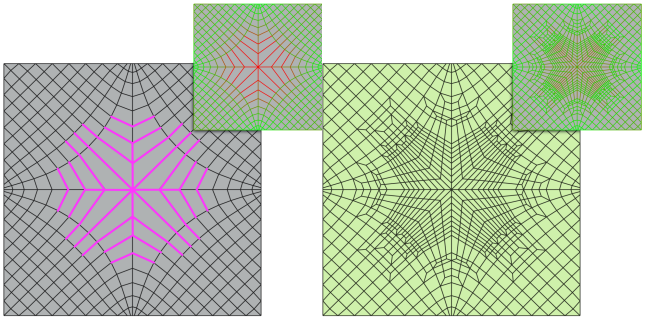


Figure 12: *Refinement of edges based on physically simulated forces acting on a roof. Top right shows the integrated stress per edge colorcoded from low (green) to high (red). After the splits the stress is distributed over more edges reducing the maximal value.*

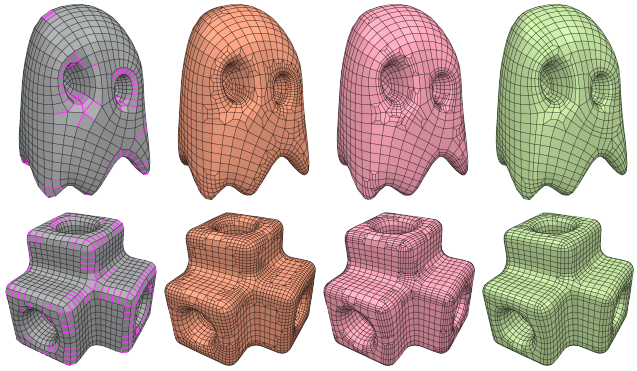


Figure 13: *Comparison of different refinement methods. From left to right: input, Schneiders [Sch96], Tchon et al. [TDC04], ours. See Table 1 for number of elements and singularities.*

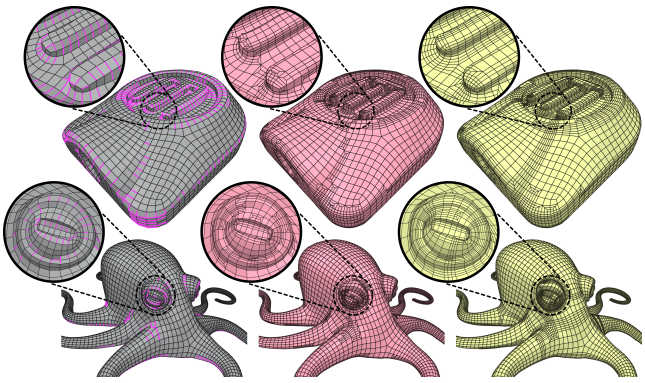


Figure 14: *Comparison of splitting the magenta edges (left) using Tchon et al. [TDC04] (middle) and our 3-refinement (right).*

ing to a mesh where the maximal integrated stress is reduced to a quarter of that of the input.

7.1. Comparison

In this section we compare our results with two other approaches for local quad mesh refinement, namely the one of Schneiders [Sch96] and the one of Tchon et al. [TDC04]. Since both methods are not applicable to applications that require control over singularity placement (cf. Figures 8 and 9) we only consider the task of improving Hausdorff distance.

Schneiders' refinement [Sch96] requires as input a refinement level per vertex. We thus set the refinement level to 1 for all vertices incident to a mandatory split edge $e \in \mathcal{S}$. The results in Figure 13 show how the isotropic refinement leads to an overrefinement of edges pointing along the smaller principal curvature direction while also inserting many singularities.

The anisotropic refinement of Tchon et al. [TDC04] is able to split faces in only one of the directions resulting in fewer elements than Schneiders. However, since Tchon et al. use 3-refinement it still produces more elements than our 2-refinement. In addition, the strictly local nature of the method does not allow splitting additional edges in order to prevent the insertion of singularities resulting in meshes with both more elements and more singularites than our method, cf. Table 1.

In Figure 14 we compare the refinment of Tchon et al. [TDC04] with our 3-refinement formulation. While the refinement of Tchon et al. will always only split the edges that are marked to be split (magenta) introducing many singularities, our formulation allows a trade-off controlled by the singularity panelty which splits additional edges but adds fewer singularities.

7.2. Limitations

The main drawback of our method is the unpredictable run time needed by the numerical solver to find the optimal solution and verify its optimality. This limitation comes from the fact that solving a binary problem is in general NP-hard.

With the Cylinder in Figure 15 (left) we found a problem that is particularly difficult for Gurobi to optimize. The set of mandatory split edges $\mathcal{S}$ consists of the 19 edges of the boundary on the left side. A constant $c_i = 1$ is chosen to prefer local refinement. 18 of the edges can be conformingly split by inserting pairs of y-

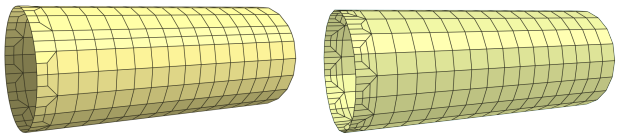


Figure 15: *Splitting the left boundary of the cylinder requires splitting one edge of the right boundary due to an odd number of edges on each boundary. With 3-refinement this is not necessary.*

configurations, for the remaining edge one of the edges on the right boundary has to be split. While Gurobi found the optimal solution in one second it took almost 10 minutes to proof optimality. To alleviate this problem Gurobi could be run only until a certain optimality gap is reached. Alternatively, our 3-refinement formulation could be used which is able to locally refine the mesh even if the boundary contains an odd number of elements, cf. Figure 15 (right).

8. Conclusions and Future Work

We presented a binary program formulation whose solution is a conforming quad mesh refinement which is optimal with respect to user preferences regarding singularities and mesh complexity. Our formulation provides detailed control over both number and location of singularities by employing user defined singularity costs.

We extended our method by providing a set of replacement templates for y-configuration, thus guaranteeing that every quad corner is split at most once, improving the minimal inner angles in the resulting mesh.

Finally, a variation of our formulation solves a simpler problem at the cost of performing 3-refinment instead of 2-refinement.

To our knowledge this is the first algorithm allowing the trade-off between mesh complexity and number of singularities as well as performing local anisotropic refinement of quad meshes.

In the future we would like to investigate further extensions to our problem formulation. Some ideas are constraining the maximal vertex valence introduced by the method, or constraining the number of edges that need to be split, e.g. in order to obtain an exact number of edges on a boundary in order to stitch it with another one.

Another interesting line of future investigation would be the development of a specialized solver that is able to solve the presented problems in a shorter and more predictable runtime, maybe at the cost of not finding the optimal solution, for applications where a good solution that is computed fast is more desirable than a provably optimal solution.

Acknowledgements

We would like to thank Jan Möbius for creating and maintaining the geometry processing framework OpenFlipper [MK12] and Juan Musto for providing the simulation results for Figure 12. Input models are provided by [MPZ14] and [ZJ16]. This work was supported by the Gottfried-Wilhelm-Leibniz Programme of the Deutsche Forschungsgemeinschaft DFG and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (AlgoHex, grant agreement No 853343). Open access funding enabled and organized by Projekt DEAL. [Correction added on 08 December 2020, after first online publication: Projekt Deal funding statement has been added.]

References

[ABO09] ANDERSON B., BENZLEY S., OWEN S.: *Automatic All Quadrilateral Mesh Adaption through Refinement and Coarsening*. 2009. doi:10.1007/978-3-642-04319-2_32. 2

[BCE*13] BOMMES D., CAMPEN M., EBKE H.-C., ALLIEZ P., KOBBELT L.: Integer-grid Maps for Reliable Quad Meshing. *ACM Trans. Graph. 32* (2013). doi:10.1145/2461912.2462014. 1

[BLK11] BOMMES D., LEMPFER T., KOBBELT L.: Global Structure Optimization of Quadrilateral Meshes. *Computer Graphics Forum 30*, 2 (2011). doi:10.1111/j.1467-8659.2011.01868.x. 2

[BLP*12] BOMMES D., LÉVY B., PIETRONI N., PUPPO E., A C. S., TARINI M., ZORIN D.: State of the art in quad meshing. In *Eurographics STARS* (2012). 1

[BZK09] BOMMES D., ZIMMER H., KOBBELT L.: Mixed-integer Quadrangulation. In *ACM SIGGRAPH 2009 Papers* (2009), SIGGRAPH '09, ACM. doi:10.1145/1576246.1531383. 1

[CAS*19] CHERCHI G., ALLIEZ P., SCATENI R., LYON M., BOMMES D.: Selective Padding for Polycube-Based Hexahedral Meshing. *Computer Graphics Forum 38*, 1 (2019). doi:10.1111/cgf.13593. 2

[CBK15] CAMPEN M., BOMMES D., KOBBELT L.: Quantized Global Parametrization. *ACM Trans. Graph. 34*, 6 (2015). doi:10.1145/2816795.2818140. 1, 6

[DSSC08] DANIELS J., SILVA C. T., SHEPHERD J., COHEN E.: Quadrilateral Mesh Simplification. In *ACM SIGGRAPH Asia 2008 Papers* (2008), SIGGRAPH Asia '08, ACM. doi:10.1145/1457515.1409101. 2

[FBT*18] FANG X., BAO H., TONG Y., DESBRUN M., HUANG J.: Quadrangulation through Morse-Parameterization Hybridization. *ACM Trans. Graph. 37*, 4 (2018). doi:10.1145/3197517.3201354. 1

[GO16] GUROBI OPTIMIZATION I.: Gurobi optimizer reference manual, 2016. 6

[HZN*18] HUANG J., ZHOU Y., NIESSNER M., SHEWCHUK J. R., GUIBAS L. J.: QuadriFlow: A Scalable and Robust Method for Quadrangulation. *Computer Graphics Forum* (2018). doi:10.1111/cgf.13498. 1

[JFH*15] JIANG T., FANG X., HUANG J., BAO H., TONG Y., DESBRUN M.: Frame Field Generation through Metric Customization. *ACM Trans. Graph. 34*, 4 (2015). doi:10.1145/2766927. 1

[JTPSH15] JAKOB W., TARINI M., PANOZZO D., SORKINE-HORNUNG O.: Instant Field-aligned Meshes. *ACM Trans. Graph. 34*, 6 (2015). doi:10.1145/2816795.2818078. 1

[KNP07] KÄLBERER F., NIESER M., POLTHIER K.: QuadCover - Surface Parameterization using Branched Coverings. *Computer Graphics Forum* (2007). doi:10.1111/j.1467-8659.2007.01060.x. 1

[Kob96] KOBBELT L.: Interpolatory Subdivision on Open Quadrilateral Nets with Arbitrary Topology. *Computer Graphics Forum 15*, 3 (1996). doi:10.1111/1467-8659.1530409. 3

[LCBK19] LYON M., CAMPEN M., BOMMES D., KOBBELT L.: Parametrization quantization with free boundaries for trimmed quad meshing. *ACM Transactions on Graphics 38*, 4 (2019). 7

[LHJ*15] LING R., HUANG J., JÜTTLER B., SUN F., BAO H., WANG W.: Spectral Quadrangulation with Feature Curve Alignment and Element Size Control. *ACM Trans. Graph. 34*, 1 (2015). doi:10.1145/2653476. 1

[MK12] MÖBIUS J., KOBBELT L.: OpenFlipper: An Open Source Geometry Processing and Rendering Framework. In *Curves and Surfaces*, vol. 6920 of *Lecture Notes in Computer Science*. 2012. 9

[MPZ14] MYLES A., PIETRONI N., ZORIN D.: Robust Field-aligned Global Parametrization. *ACM Trans. Graph. 33*, 4 (2014). doi:10.1145/2601097.2601154. 1, 9

[NHE*19] NUVOLI S., HERNANDEZ A., ESPERANÇA C., SCATENI R., CIGNONI P., PIETRONI N.: QuadMixer: Layout Preserving Blending of Quadrilateral Meshes. *ACM Trans. Graph. 38*, 6 (2019). doi:10.1145/3355089.3356542. 2

[NW88] NEMHAUSER G. L., WOLSEY L. A.: *Integer and Combinatorial Optimization*. Wiley-Interscience, 1988. 3

[PPTSH14] PANOZZO D., PUPPO E., TARINI M., SORKINE-HORNUNG O.: Frame Fields: Anisotropic and Non-Orthogonal Cross Fields. *ACM Trans. Graph. 33*, 4 (2014). doi:10.1145/2601097.2601179. 1

[PZKW11] PENG C.-H., ZHANG E., KOBAYASHI Y., WONKA P.: Connectivity Editing for Quadrilateral Meshes. In *Proceedings of the 2011 SIGGRAPH Asia Conference* (2011), SA '11, ACM. doi:10.1145/2024156.2024175. 2

[Sch96] SCHNEIDERS R.: Refining Quadrilateral and Hexahedral Element Meshes. In *5th International Conference on Grid Generation in Computational Field Simulations* (1996), CRC Press. 2, 4, 6, 8

[TDC04] TCHON K.-F., DOMPIERRE J., CAMARERO R.: Automated refinement of conformal quadrilateral and hexahedral meshes. *International Journal for Numerical Methods in Engineering 59* (2004). doi:10.1002/nme.926. 2, 6, 8

[TPC*10] TARINI M., PIETRONI N., CIGNONI P., PANOZZO D., PUPPO E.: Practical quad mesh simplification. *Computer Graphics Forum 29*, 2 (2010). doi:10.1111/j.1467-8659.2009.01610.x. 2

[VS16] VERMA C. S., SURESH K.: αMST: A robust unified algorithm for quadrilateral mesh adaptation. *Procedia Engineering 163* (2016). doi:https://doi.org/10.1016/j.proeng.2016.11.053. 2

[ZHLB10] ZHANG M., HUANG J., LIU X., BAO H.: A Wave-Based Anisotropic Quadrangulation Method. *ACM Trans. Graph. 29*, 4 (2010). doi:10.1145/1778765.1778855. 1

[ZHLB13] ZHANG M., HUANG J., LIU X., BAO H.: A Divide-and-Conquer Approach to Quad Remeshing. *IEEE Transactions on Visualization and Computer Graphics 19*, 6 (2013). 1

[ZJ16] ZHOU Q., JACOBSON A.: Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016). 9

Appendix

Indicator Variables z_i^j

First, we show that the boolean expression

$$z \Leftrightarrow \bigwedge_{i=1}^{n} x_i \wedge \bigwedge_{j=1}^{m} \neg y_j \tag{15}$$

is equivalent to the linear constraints

$$0 \leq \sum_{i=1}^{n} x_i - \sum_{j=1}^{m} y_j - (n+m)z + m \leq n+m-1. \tag{16}$$

Proof:

"⇒": If $z = 1$ then $0 \leq \sum_{i=1}^{n} x_i - \sum_{j=1}^{m} y_j - n$ can only be true if all $x_i = 1$ and all $y_j = 0$.

"⇐": If $z = 0$ then $\sum_{i=1}^{n} x_i - \sum_{j=1}^{m} y_j \leq n-1$ can only be false if all $x_i = 1$ and all $y_j = 0$. Thus at least one $x_i = 0$ or one $y_j = 1$. □

With this it is easy to express that the variables z_i^j indicating which template from Figure 5 is used to replace a y-configuration are 1 if and only if these edges are split and those are not. We simply need to use the above formula and replace the x_i with the s_i of those edges that are split and the y_i with the ones not being split. This leads to the following 16 constraints per y-configuration (left column of Figure 5 first):

$$\begin{aligned}
0 &\leq s_0+s_1-s_2-s_3-s_4-s_5+s_6-s_7-s_8+6-9z_i^1 \leq 8\\
0 &\leq s_0+s_1+s_2+s_3-s_4-s_5+s_6+s_7-s_8+3-9z_i^2 \leq 8\\
0 &\leq s_0+s_1+s_2-s_3+s_4-s_5+s_6+s_7-s_8+3-9z_i^3 \leq 8\\
0 &\leq s_0+s_1-s_2+s_3+s_4-s_5+s_6-s_7-s_8+4-9z_i^4 \leq 8\\
0 &\leq s_0+s_1-s_2-s_3-s_4-s_5-s_6+s_7+s_8+5-9z_i^5 \leq 8\\
0 &\leq s_0+s_1+s_2+s_3-s_4-s_5-s_6-s_7+s_8+4-9z_i^6 \leq 8\\
0 &\leq s_0+s_1+s_2-s_3+s_4-s_5-s_6-s_7+s_8+4-9z_i^7 \leq 8\\
0 &\leq s_0+s_1-s_2+s_3+s_4-s_5-s_6+s_7+s_8+3-9z_i^8 \leq 8\\
0 &\leq s_0+s_1+s_2-s_3-s_4+s_5-s_6-s_7-s_8+5-9z_i^9 \leq 8\\
0 &\leq s_0+s_1-s_2+s_3-s_4+s_5-s_6+s_7-s_8+4-9z_i^{10} \leq 8\\
0 &\leq s_0+s_1-s_2-s_3+s_4+s_5-s_6+s_7-s_8+4-9z_i^{11} \leq 8\\
0 &\leq s_0+s_1+s_2+s_3+s_4+s_5-s_6-s_7-s_8+3-9z_i^{12} \leq 8\\
0 &\leq s_0+s_1+s_2-s_3-s_4+s_5+s_6+s_7+s_8+2-9z_i^{13} \leq 8\\
0 &\leq s_0+s_1-s_2+s_3-s_4+s_5+s_6-s_7+s_8+3-9z_i^{14} \leq 8\\
0 &\leq s_0+s_1-s_2-s_3+s_4+s_5+s_6-s_7+s_8+3-9z_i^{15} \leq 8\\
0 &\leq s_0+s_1+s_2+s_3+s_4+s_5+s_6+s_7+s_8+0-9z_i^{16} \leq 8
\end{aligned}$$

8.1. Y-Configuration Refinement Costs d_i^j

In Section 5.1 we discussed that the standard energy penalizes the refinement of y-configuration too much, thus we added an adjustment energy E_a (Equation (10)). Here we give the correct compensation costs d_i^j for all y-configuration refinements for the case of a constant singularity penalty $c_i = c$.

To compute the correct compensation d_i^j we compare the cost of creating the yellow configuration in Figure 5, i.e. the cost of two splits and a y-configuration to create the initial faces plus the number of splits and inserted y-configuration depicted in the figure, with the cost of first refining regularily (four splits) and then inserting the y-configurations for the green configurations. The difference between these costs needs to be accounted for by d_i^j. Thus, we get the following costs:

$$\begin{aligned}
d_i^1 &= -3c-1 & d_i^9 &= -2c+2\\
d_i^2 &= -c-1 & d_i^{10} &= 2\\
d_i^3 &= -c-1 & d_i^{11} &= 2\\
d_i^4 &= -3c-1 & d_i^{12} &= -2c+2\\
d_i^5 &= 2c+2 & d_i^{13} &= c-1\\
d_i^6 &= 2 & d_i^{14} &= -c-1\\
d_i^7 &= 2 & d_i^{15} &= -c-1\\
d_i^8 &= 2c+2 & d_i^{16} &= c-1
\end{aligned}$$